As-if Infinitely Ranged Integer Model

David Keaton
Thomas Plum
Robert C. Seacord
David Svoboda
Alex Volkovitsky
Timothy Wilson

July 2009

TECHNICAL NOTE
CMU/SEI-2009-TN-023

CERT Program

http://www.sei.cmu.edu

Carnegie Mellon

This report was prepared for the

SEI Administrative Agent
ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2100

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications section of our website (http://www.sei.cmu.edu/publications/).

Table of Contents

Acknowledgments **iii**

Abstract **iv**

1 Integral Security **1**
- 1.1 Signed Integer Overflow 1
- 1.2 Unsigned Integer Wrapping 1
- 1.3 Conversion Errors 2
- 1.4 Optimizations that Exploit Undefined Behavior 3

2 Alternative Solutions **5**
- 2.1 The GCC –ftrapv Flag 5
- 2.2 Precondition Testing 5
- 2.3 Saturation Semantics 7
- 2.4 Overflow Detection 8
- 2.5 Branch Insertion 9
- 2.6 Runtime Integer Checking (RICH) 10
- 2.7 Clang Implementation 10
- 2.8 GCC no-undefined-overflow 11

3 As-if Infinitely Ranged (AIR) Integers **12**
- 3.1 Implementation Methods 13
- 3.2 Undefined Behavior 14
  - 3.2.1 Division by 0, modulo 0 (signed or unsigned) 15
  - 3.2.2 INT_MIN / -1 15
  - 3.2.3 INT_MIN % -1 15
  - 3.2.4 Shifts 15
  - 3.2.5 Enabling and Disabling Unsigned Integer Wrapping 16
  - 3.2.6 Integer Promotions and the Usual Arithmetic Conversions 17
  - 3.2.7 Integer Constants 17
  - 3.2.8 Expressions Involving Integer Variables and Constants 18
  - 3.2.9 Runtime-constraint Handling 18
- 3.3 Array Dereferences 19
- 3.4 The rsize_t Type 19
- 3.5 Fussy Overflows 19
- 3.6 Optimizations 20
- 3.7 GCC Prototype 21
- 3.8 Experimental Results 22

Summary and Conclusions **25**

References/Bibliography **26**

List of Tables

Table 1. Saturation semantics 7

Table 2. SPECINT2006 macro-benchmark runs 9

Table 3. Critical undefined behavior 13

Table 4. SPECINT2006 macro-benchmark runs 22

Acknowledgments

We would like to acknowledge the contributions of the following individuals to the research presented in this paper: Ian Lance Taylor, Richard Guenther, David Chisnall, Tzi-cker Chiueh, Huijia Lin, Joseph Myers, Alexey Smirnov, Rob Johnson, and David Brumley. We would also like to acknowledge the contribution of our technical editors, Edward Desautels and Alexa Huth.

Abstract

Integer overflow and wraparound are major causes of software vulnerabilities in the C and C++ programming languages. In this paper we present the as-if infinitely ranged (AIR) integer model, which provides a largely automated mechanism for eliminating integer overflow and integer truncation. The AIR integer model either produces a value equivalent to one that would have been obtained using infinitely ranged integers or results in a runtime constraint violation. Unlike previous integer models, AIR integers do not require precise traps, and consequently do not break or inhibit most existing optimizations.

1 Integral Security

The majority of software vulnerabilities result from coding errors. For example, 64% of the vulnerabilities in the National Vulnerability Database in 2004 resulted from programming errors [Heffley 04].[1] The C and C++ languages are particularly prone to coding errors that result in vulnerabilities because of the lack of type-safety in these languages [Seacord 2005].

In 2007, MITRE reported that buffer overflows remain the number one issue as reported in operating system (OS) vendor advisories. It also reported that integer overflow, barely in the top 10 overall in the years preceding the report, was number two for OS vendor advisories [Christey 07].

Integer values that originate from untrusted sources and are used in any of the following ways can easily result in vulnerabilities:

- as an array index
- in any pointer arithmetic
- as a length or size of an object
- as the bound of an array (for example, a loop counter)
- as an argument to a memory allocation function

The following subsections describe specific integer behaviors that are known to result in vulnerabilities in real-world applications.

1.1 Signed Integer Overflow

Signed integer overflow is undefined behavior in C, allowing implementations to silently wrap (the most common behavior), trap, or both. Because signed integer overflow produces a silent wraparound in most existing C and C++ implementations, some programmers assume that this is a well-defined behavior. However, modern compilers may assume during optimization that undefined behaviors cannot occur, and eliminate code that can only execute in the presence of undefined behavior.

Rule INT32-C in The CERT C Secure Coding Standard requires programmers to "Ensure that operations on signed integers do not result in overflow" [Seacord 2008].

1.2 Unsigned Integer Wrapping

Although unsigned integer wrapping is well-defined by the C standard as having modulo behavior, unexpected wrapping has led to numerous software vulnerabilities. The following code shows an example of an integer overflow based on a real-world vulnerability in the handling of the comment field in JPEG files [Solar Designer 00].

[1] Other broad categories of vulnerabilities include design and configuration errors.

```
void getComment(size_t len, char *src) {
  size_t size;
  size = len - 2;
  char *comment = (char *)malloc(size + 1);
  memcpy(comment, src, size);
  return;
}
int main(void) {
  getComment(1, "shellcode");
  return 0;
}
```

JPEG files contain a comment field that includes a two-byte length field. The length field indicates the length of the comment, including the length field. To determine the length of the comment string alone (for memory allocation), the function reads the value in the length field and subtracts two. The function then allocates the length of the comment plus one byte for the terminating null byte. There is no error check to ensure that the length field is valid, which makes it possible to cause an overflow by creating an image with a comment length field containing the value 1. The memory allocation call of zero bytes (1 minus 2 [length field] plus 1 [null termination]) succeeds. (It is implementation-defined behavior whether the `calloc()`, `malloc()`, and `realloc()` functions return a null pointer or a pointer to an allocated object when the size requested is zero.) The size variable is declared as a size_t (line 2), resulting in a large positive value of `0xFFFFFFFF` (from 1 minus 2). Subsequently, `size + 1` evaluates to zero.

Another real-world example of vulnerabilities resulting from unsigned integer wrapping occurs in memory allocation. Wrapping can occur in `calloc()` and other memory allocation functions when computing the size of a memory region. As a result, a buffer is returned that is smaller than the requested size, possibly resulting in a subsequent buffer overflow.[2]

The following code fragments may lead to wrapping vulnerabilities, where `count` is an unsigned integer:

```
C: pointer = calloc(sizeof(element_t), count);
C++: pointer = new ElementType[count];
```

This secure coding rule is captured by INT30-C, "Ensure that unsigned integer operations do not wrap" in the CERT C Secure Coding Standard [Seacord 2008].

1.3 Conversion Errors

The CERT C Secure Coding Standard rule INT31-C requires that integer conversions, both implicit and explicit (using a cast), must be guaranteed not to result in lost or misinterpreted data [Seacord 2008].

The only integer type conversions that are guaranteed to be safe for all data values and all possible conforming implementations are conversions of an integral value to a wider type of the same signedness. Conversion of an integer to a smaller type results in truncation of the high-order bits.

[2] See http://cert.uni-stuttgart.de/advisories/calloc.php

Consequently, assigning an integer with greater precision to an integer type with lesser precision can result in truncation if the resulting value cannot be represented in the smaller type.

The following code illustrates a truncation vulnerability from the SSH CRC-32 Compensation Attack Detector [CVE 2001].

```
int detect_attack(u_char *buf, int len, u_char *IV) {
  static word16 *h = (word16 *)NULL;
  static word16 n = HASH_MIN_ENTRIES;
  word32 l;
  ...
  for (l = n; l < HASH_FACTOR(len/BSIZE); l = l << 2);
  if (h == NULL) {
    debug("Install crc attack detector.");
    n = l;
    h = (word16 *) xmalloc(n * sizeof(word16));
  } else
  for (c = buf, j = 0; c < (buf+len); c += BSIZE, j++){
    for (i = HASH(c) & (n ? 1); h[i] != UNUSED;
      i = (i + 1) & (n ? 1)) ...;
    h[i] = j;
  }
}
```

The local variable `n` is 16-bits long, so the assignment `n = l` may result in truncation. By sending a large SSH protocol packet an attacker can force this truncation to occur, causing the call to `xmalloc()` to allocate too little space. The code that initializes the allocated space a few lines later will corrupt SSH's memory, leading to an attack.

1.4 Optimizations that Exploit Undefined Behavior

Conforming implementations can deal with undefined behavior in a variety of fashions, such as ignoring the situation completely, with unpredictable results; translating or executing the program in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message); or terminating a translation or execution (with the issuance of a diagnostic message). Because compilers are not obligated to generate code for undefined behavior, these behaviors are candidates for optimization. By assuming that undefined behaviors will not occur, compilers can generate code with better performance characteristics.

Signed integer overflow is often not considered a problem for hardware which detects it, because overflow is undefined behavior and a program is permitted to exhibit any behavior once undefined behavior has occurred.

For example, this code tests for signed integer overflow by testing if `a + 100 > a`:

```
#include <assert.h>

int foo(int a) {
  assert(a + 100 > a);
  printf("%d %d\n", a + 100, a);
```

```
  return a;
}

int main(void) {
  foo(100);
  foo(INT_MAX);
}
```

This test cannot evaluate to false unless an integer overflow occurs. However, because this code depends on undefined behavior, a compiler may eliminate it. For example, GCC version 4.1.1 optimizes out the assertion for all optimization levels, and GCC 4.2.3 optimizes out the assertion for programs compiled with `-O2` level optimization and higher.

If an implementation traps on overflow, the integer overflow will cause the program to terminate (before it has an opportunity to perform the test).

In the following example, derived from the GNU C Library 2.5 implementation of `mktime` (2006-09-09), the code assumes wraparound arithmetic in the addition operation to detect signed overflow:

```
  time_t t, t1, t2;
  int sec_requested, sec_adjustment;
    ...
  t1 = t + sec_requested;
  t2 = t1 + sec_adjustment;
  if (((t1 < t) != (sec_requested < 0))
    | ((t2 < t1) != (sec_adjustment < 0)))
    return -1;
```

Although it is often assumed that a C compiler must honor the presence of parentheses, even for commutative-associative operators, this is not really true. The compiler can follow the "as if" rule and rearrange expressions provided the correct result is computed. It is usually true that the compiler is prevented from rearranging floating point expressions, but a compiler can often rearrange integral expressions, which may cause an overflow condition. Historically, this has not been considered to be a problem.

All of this puts the onus on the programmer to write strictly conforming code, with or without the help of the compiler.

2 Alternative Solutions

This section describes existing and theoretical alternative solutions to the problem of integral security in C and explains why these solutions inadequately address the issues.

2.1 The GCC –ftrapv Flag

GCC provides an `-ftrapv` compiler option that provides limited support for detecting integer overflows at runtime. The GCC runtime system generates traps for signed overflow on addition, subtraction, and multiplication operations for programs compiled with the `-ftrapv` flag. This is accomplished by invoking existing, portable library functions that test an operation's post-conditions and call the C library `abort()` function when results indicate that an integer error has occurred [Seacord 2005].

The following function from the GCC runtime system is used to detect overflows resulting from the addition of signed 16-bit integers.

```
Wtype __addvsi3(Wtype a, Wtype b) {
   const Wtype w = a + b;
   if (b >= 0 ? w < a : w > a)
     abort ();
   return w;
}
```

The two operands are added and the result is compared to the operands to determine whether an overflow condition has occurred. For `_addvsi3()`, if `b` is non-negative and `w < a`, an overflow has occurred and `abort()` is called. Similarly, `abort()` is called if b is negative and `w > a`.

The GCC `-ftrapv` flag only works for a limited subset of signed operations and always results in an `abort()` when a runtime overflow is detected.

The `-ftrapv` option is known to have substantial problems, and discussions for how to trap signed integer overflows in a reliable and maintainable manner are on-going within the GCC community.

2.2 Precondition Testing

Another approach to eliminating integer exceptional conditions is to test the values of the operands before an operation to prevent overflow and wraparound from occurring. This is especially important for signed integer overflow, which is undefined behavior and may result in a trap on some architectures (for example, a division error on IA-32). The following code illustrates a strictly conforming test to ensure that the ensuing multiplication operation does not result in an overflow.

```
signed int si1, si2, result;

/* Initialize si1 and si2 */

if (si1 > 0){  /* si1 is positive */
  if (si2 > 0) {  /* si1 and si2 are positive */
    if (si1 > (INT_MAX / si2)) {
      /* handle error condition */
    }
  } /* end if si1 and si2 are positive */
  else { /* si1 positive, si2 non-positive */
    if (si2 < (INT_MIN / si1)) {
        /* handle error condition */
    }
  } /* si1 positive, si2 non-positive */
} /* end if si1 is positive */
else { /* si1 is non-positive */
  if (si2 > 0) { /* si1 is non-positive, si2 is positive */
    if (si1 < (INT_MIN / si2)) {
      /* handle error condition */
    }
  } /* end if si1 is non-positive, si2 is positive */
  else { /* si1 and si2 are non-positive */
    if ( (si1 != 0) && (si2 < (INT_MAX / si1))) {
      /* handle error condition */
    }
  } /* end if si1 and si2 are non-positive */
} /* end if si1 is non-positive */

result = si1 * si2;
```

Similar examples of precondition testing are shown in The CERT C Secure Coding Guidelines: INT30-C, “Ensure that unsigned integer operations do not wrap”; INT31-C, “Ensure that integer conversions do not result in lost or misinterpreted data”; and INT32-C, “Ensure that operations on signed integers do not result in overflow.”

Detecting an overflow in this manner can be relatively expensive, especially if the code is strictly conforming. Frequently, these checks must be in place before suspect system calls that may or may not perform similar checks before performing integral operations. Redundant testing by caller and by called is a style of defensive programming that has been largely discredited within the C and C++ community. The usual discipline in C and C++ is to require validation only on one side of each interface.

Furthermore, branches can be expensive on modern hardware, so programmers and implementers work hard to keep branches out of inner loops. This argues against requiring the application programmer to pre-test all arithmetic values to prevent an overflow, which rarely happens. Preventing run-time overflow by program logic is sometimes easy, sometimes complicated, and sometimes extremely difficult. Clearly, some overflow occurrences can be diagnosed in advance by static-analysis methods. But no matter how good this analysis is, there are still code sequences

that cannot be detected before run time. In most cases, the resulting code is much less efficient than what a compiler could generate to recognize that an overflow took place.

The underlying process of code generation may be immensely complicated, but in general it is best to avoid complexity in the code that end-user programmers are required to write.

2.3 Saturation Semantics

Verifiably in-range operations are often preferable to treating out-of-range values as an error condition because the handling of these errors has been shown to cause denial-of-service problems in actual applications. The quintessential example of this is the failure of the Ariane 5 launcher, which occurred because of an improperly handled conversion error that resulted in the processor being shut down [Lions 96].

A program that detects an imminent integer overflow may either signal an error condition or produce an integer result that is within the range of representable integers on that system. Some applications, particularly in embedded systems, are better handled by producing a verifiably in-range result because it allows the computation to proceed, thereby avoiding a denial-of-service attack. However, when continuing to produce an integer result in the face of overflow, the question of what integer result to return to the user must be considered.

The saturation and modwrap algorithms and the technique of restricted range usage produce integer results that are always within a defined range. This range is between the integer values `MIN` and `MAX` (inclusive), where `MIN` and `MAX` are two representable integers with `MIN < MAX`.

For saturation semantics, assume that the mathematical result of the computation is `result`. The value actually returned to the user is set out in the following table:

Table 1. Saturation semantics

Range of mathematical result	Result returned
`MAX < result`	`MAX`
`MIN <= result <= MAX`	`result`
`result < MIN`	`MIN`

In the C standard, signed integer overflow produces undefined behavior, meaning that any behavior is permitted. Consequently, producing a saturated `MAX` result is permissible. The implementation of saturation semantics for unsigned integers would require a change in the standard. For both signed and unsigned integers, there is currently no way of *requiring* a saturated result. If C1X defined a new standard pragma such as `_Pragma(STDC SAT)`, saturation semantics could be provided without impacting existing code.

Although saturation semantics may be suitable for some applications, it is not always appropriate in security critical code where abnormal integer values may be an indication of an attack.

2.4 Overflow Detection

C99 provides the `<fenv.h>` header to support the floating-point exception status flags and directed-rounding control modes required by IEC 60559, and other similar floating-point state information. This includes the ability to determine which floating-point exception flags are set.

It is ironic that floating-point has a set of fully developed methods for monitoring and reporting exceptional conditions, even though the population using those methods is orders of magnitude smaller than the population that needs correctly represented integers. On the other hand, perhaps C's long gestation period for addressing the correct-representation problem will lead to a system which is superior to the other languages that tackled the problem decades ago (such as Pascal and Ada).

A potential solution to handling integer exceptions in C is to provide an inquiry function (just as C provides for floating point) that interrogates status flags that are being maintained by the (compiler-specific) assembler code that performs the various integer operations. If the inquiry function is called after an integral operation and returns a "no overflow" status, then the value is reliably correctly represented.

At the level of assembler code, the cost of detecting overflow is zero or nearly zero. Many architectures do not even have an instruction for "add two numbers but do NOT set the overflow or carry bit"; the detection occurs for free whether it is desired or not. But it is only the specific compiler code generator that knows what to do with those status flags.

These inquiry functions may be defined, for example, by translating the `<fenv.h>` header into an equivalent `<ienv.h>` header that provides access to the integer exception environment. This header would support the integer exception status flags, and other similar integer exception state information.

However, anything that can be performed by an `<ienv.h>` interface could be better performed by the compiler. For example, the compiler may choose a single, cumulative integer exception flag in some cases, and one flag per variable in others, depending on what is most efficient in terms of speed and storage for the particular expressions involved. Additionally, the concept of a runtime constraint handler did not exist until the publication of TR 24731-1 [ISO/IEC TR 24731-1:2007], so the C standards committee defined an interface that put the entire burden on the programmer.

Floating-point code is different from integer code in that it includes concepts such as rounding mode, which need not be considered for integers. Additionally, floating point has a specific value, NaN (Not a Number), which indicates that an unrepresentable value was generated by an expression. Sometimes floating-point programmers want to terminate a computation when a NaN is generated; at other times they want to print out the NaN because its existence conveys valuable information (and there might be one NaN in the middle of an array being printed out, with the rest of the values being valid results). Because of the combination of NaNs and the lack of runtime constraint handlers, the programmer needed to be given more control.

In general, there is no NaI (Not an Integer) value, so there is no requirement to preserve such a value to allow it to be printed out. Therefore, the programmer does not need fine control over whether or not an integer runtime constraint handler gets called after each operation. Without this requirement, it is preferable to keep the code simple and let the compiler do the work, which it can generally do more reliably and efficiently than individual application programmers.

2.5 Branch Insertion

Branch insertion is a proof of concept modification to the GCC Compiler, developed by CERT, which automatically invokes a runtime constraint handler when an integer operation fails to produce a correctly represented value.

An x86 processor has two flags that deal with integer overflow, a carry flag, which is set when the most significant bit changes from a 1 to a 0, and an overflow flag, which is set when the most significant bit changes from a 0 to a 1. On x86 processors, the simplest way to detect integer overflow is with instructions that branch conditionally on the status of the carry and overflow flags as follows:

```
   // arithmetic
   jn[co] .LANALYZEXXX
   call overflow_handler
.LANALYZEXXX
   // code after arithmetic
```

Because signedness information is required to determine which of the overflow or carry flags to branch on, and because GCC's GIMPLE representation is the final representation which retains signedness information, GCC's GIMPLE translation stage was modified to insert additional assembly code after arithmetic operations. This was accomplished by inserting an additional `GIMPLE_ASM` statement following any `GIMPLE_ASSIGN` statement which involved a computation that could potentially overflow such as `PLUS_EXPR`, `MINUS_EXPR`, `MULT_EXPR`, `NEGATE_EXPR`, and `LSHIFT_EXPR`.

The performance of branch insertion was assessed using the SPECINT2006 benchmarks, composed of several open source programs. SPECINT2006 was compiled using a reference (unmodified) GCC compiler and a GCC compiler modified to implement branch insertion. The two binaries were then run, and the ratio of their runtime to a known baseline was used to compute a performance ratio.

The higher numbers in Table 2 indicate better performance.

Table 2. SPECINT2006 macro-benchmark runs

Optimization Level	Control Ratio	Analyzable Ratio	% Slowdown
-O0	4.93	4.67	5.5%
-O1	7.13	6.30	13%
-O2	7.45	6.59	13%

As expected, for non-optimized code, the effective slowdown was on the order of around 5%. This represents the best case performance due entirely to the addition of jump statements following arithmetic operations.

The slowdown in optimized code (`-O1` and `-O2`) occurs because GCC only performs optimizations on basic blocks. However, if a branch is inserted into the code sequence then it splits the code so that there are now more and smaller basic blocks. This leaves less room for local optimizations (intra-basic-block) to do their job, and makes more work for global optimizations (inter-basic-block) as well as making functions appear larger (more basic blocks) to inter-procedural optimizations such as inlining, perhaps inhibiting some inlining that was done before.

Because the overflow branch should never be taken except in cases involving unexpected behavior, the branch predictor can remember whether or not the jump was taken, and thereby make a correct prediction in all subsequent encounters. Because, in the worst case, the branch predictor will mispredict once per jump, and the number of jumps is fixed at compile time, the number of mispredicted jumps is constant.

2.6 Runtime Integer Checking (RICH)

Brumley et al. have developed a static program transformation tool, called RICH, that takes as input any C program and outputs object code that monitors its own execution to detect integer overflows and other bugs [Brumley 2007]. Despite the ubiquity of integer operations, the runtime performance penalty of RICH is low, averaging less than 5%. RICH implements the checks in two phases. At compile time, RICH instruments the target program with run-time checks of all unsafe integer operations. At run time, the inserted instrumentation checks each integer operation. When a check detects an integer error, it generates a warning and optionally terminates the program.

2.7 Clang Implementation

David Chisnall implemented the AIR integer model for Clang using the LLVM overflow-checked operations.[3] The current implementation checks the integer overflow flag after each `+`, `-` or `*` integer operation and, in the case of an overflow, calls a handler function such as:

```
extern long long (*__overflow_handler)(
  long long a, long long b, char op, char  width
);
```

The operation arguments are promoted to the `long long` type via sign extension, the op indicates whether it was signed/unsigned addition, subtraction or multiplication, and the width indicates the expected width of the result. GCC's `-ftrapv` can be replicated by having it unconditionally call `abort()`. Alternatively, overflow can be handled by calling a registered handler function from a stack, or by promoting it to some kind of boxed value. If this function returns, its return value is truncated and used in place of the result of the operation.

The Clang implementation simply checks the flag immediately after any signed integer operation and jumps to a handler function if overflow occurred. Conditional jumps on overflow are cheap

[3] http://article.gmane.org/gmane.comp.compilers.clang.devel/4469

because the branch predictor will almost always guess correctly. By allowing a custom handler function, rather than abort, Clang allows for calling `longjmp()` or some unwind library functions in cases where overflow occurred. This works well with the optimizer, which can eliminate the test for cases where the value can be proven to be in-range.

2.8 GCC no-undefined-overflow

Richard Guenther has proposed a new no-undefined-overflow branch for GCC the goal of which is to make overflow behavior explicit per operation and to eliminate all constructs in the GIMPLE intermediate language (IL) that invoke undefined behavior [Guenther 2009]. To support languages, such as C and C++, that have undefined semantics on overflowing operations, the middle-end gets new unary and binary operators that implicitly encode value-range information about their operands noting that the operation does not overflow. These does-not-overflow operators transform the undefined behavior into a valid assumption making the GIMPLE IL fully defined. Consequently, the front-end and value-range analysis must determine if operations overflow and generate the appropriate IL.

Instructions such as `NEGATE_EXPR, PLUS_EXPR, MINUS_EXPR, MULT_EXPR and POINTER_PLUS_EXPR` would have wrapping, no-overflow, and trapping variants.

The trapping variants are indicated by a `V` for overflow (for example, `PLUSV_EXPR` is the trapping variant for `PLUS_EXPR`) and by `NV` for no overflow (for example, `PLUSNV_EXPR`). The no-overflow variant also wraps if it overflows so that existing code continues to function.

The GCC no-undefined-overflow branch, when implemented, should greatly facilitate the implementation of the AIR integer model within GCC.

3 As-if Infinitely Ranged (AIR) Integers

The purpose of the AIR integer model is to either a) produce a value which is equivalent to a value that would have been obtained using infinitely ranged integers, or b) result in a runtime constraint violation. This model is generally applied to both signed and unsigned integers, although either may be enabled or disabled for entire compilation units using compiler options or locally using attributes. Implementations must declare they are implementing analyzability with a predefine, for example, `__STDC_ANALYZABLE__`. In our prototype, the AIR integer model is enabled using the GCC `-fanalyzable` compile time option, although the exact mechanism used is implementation defined. In analyzable mode, fetching an integer trap representation should not execute a trap. This should already be true on most or all existing platforms. Optimizations are encouraged provided that overflow is set properly.

The AIR integer model distinguishes between undefined behavior specified by the standard, a compiler defect, or a hardware defect. The C standards committee has always assumed that, excluding floating point, it is possible to write portable (strictly conforming) C, which requires knowledge of specific restrictions.

In the AIR integer model, when an *observation point* is reached, if overflow traps have not been disabled, and if no traps have been raised, then any integer value in the output is correctly represented ("as if infinitely ranged"). These traps are implemented using either the existing hardware traps (such as divide-by-zero) or by invoking a runtime-constraint handler. The same solution is applied to unsigned integer wrapping. Whether a program traps for given inputs depends on the exact optimizations carried out by a particular compiler version. If required, a programmer can implement a custom runtime-constraint handler to set a flag and continue (using the indeterminate value that was produced). In the future, an implementation that also supports C++ might throw an exception rather than invoke a runtime-constraint handler; or alternatively, the runtime-constraint handler can throw an exception. We have not attempted to evaluate these, or other, alternatives for C++.

An observation point occurs at an output, including volatile object accesses, or at a pointer dereference that would cause undefined behavior, or at a call to a library routine using arguments that would cause undefined behavior.

AIR Integers do not *require* Ada-style precise traps, which require that an exception is raised every time there is an integer overflow. In the AIR integer model, it is acceptable to delay catching an incorrectly represented value until an observation point is reached just before it either affects the output or causes *a critical undefined behavior* [Plum 09]. This model improves the ability of compilers to optimize, without sacrificing safety and security.

Critical undefined behavior is a means of differentiating between behaviors for which, upon use of a nonportable or erroneous program construct or of erroneous data, the C standard imposes no requirements. The behavior might, for example, perform an out-of-bounds store or a trap.

By contrast, *bounded undefined behavior* is behavior for which, upon use of a nonportable or erroneous program construct, or of erroneous data, no requirement is imposed except that the behavior cannot perform an out-of-bounds store, and that all values produced or stored are indeterminate values. The behavior might, for example, perform a trap.

In the analyzable model for AIR Integers, all undefined behaviors defined in clauses 1 through 7 of the C99 standard are (in this AIR integer model) restricted to the semantics of bounded undefined behavior, except for the critical undefined behaviors shown in Table 3.

Table 3. Critical undefined behavior

C99 Section	Proposed Critical Undefined Behavior
6.2.4	An object is referred to outside of its lifetime
6.3.2.1	An lvalue does not designate an object when evaluated
6.3.2.3	A pointer is used to call a function whose type is not compatible with the pointed-to type
6.5.3.2	The operand of the unary * operator has an invalid value
6.5.6	Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary * operator that is evaluated
7.1.4	An invalid argument to a function can cause an invalid address computation and/or invalid access
7.20.3	The value of a pointer that refers to space deallocated by a call to the free or realloc function is used
7.21.1, 7.24.4	A string or wide string utility function is instructed to access an array beyond the end of an object

All a programmer needs to do is specify which runtime-constraint handlers should be called and let the compiler do the rest. This makes programming much simpler, easier to read, and easier to maintain, which improves security.

3.1 Implementation Methods

AIR permits a wide range of implementation methods, some of which might apply to different environments and implementations:

- Overflow can set a flag which compiler-generated code will test later.
- Overflow can immediately invoke a runtime-constraint handler.
- The testing of flags can be performed at an early point (such as within the same full-expression), or delayed (subject to some restrictions).

For example, in the following code:

```
i = 32767 + 1;
j = i * 3;
if (m < 0)
  a[i] = . . .;
```

the variable `j` does not need to be checked within this code fragment (but may need to be checked later), and `i` does not need to be checked unless and until the `a[i]` expression is evaluated, but must be checked by then.

Compilers may choose a single, cumulative integer exception flag in some cases, and one flag per variable in others, depending on what is most efficient in terms of speed and storage for the particular expressions involved. For example, in the following code:

```
x++;
y++;
z++;
printf("%d", x);
```

the call to `printf()` is an observation point for the variable `x`. Any of the operations `x++`, `y++`, or `z++` can result in an overflow. Consequently, it is necessary to test the value of this overflow flag prior to the observation point (the call to `printf())` and invoke the runtime-constraint handler if the overflow flag is set:

```
/* compiler clears integer exception flags here */

x++;
y++;
z++;

if (/* integer exception flags are set */)
  runtime_constraint_handler();
printf("%d", x);
```

If only a single overflow flag is used, then one or more of these lvalues contains an incorrectly represented value, but we don't know which one. Consequently, the runtime-constraint handler will be invoked if any of the increment operations resulted in an overflow. In this case, it may be preferable for the compiler to generate a separate overflow flag for `x` so that the runtime-constraint handler need only be invoked if `x++` overflows.

Portably, a programmer can only assume, if the code reaches an observation point without invoking a runtime-constraint handler, that all the integer values are correctly represented. If a runtime constraint error occurs, all integer values that have been modified since the last observation point contain indeterminate values. In cases where the programmer wants to rerun the calculation using a higher or arbitrary-precision integer, the programmer would need to recalculate the values for all indeterminate values.

Ideally, while we would like to eliminate implementation-defined behavior in the AIR integer model, it is necessary to provide sufficient latitude for compiler implementers to perform certain optimizations.

3.2 Undefined Behavior

One of the goals of the AIR integer model is to eliminate previously undefined behaviors by providing definitions. The goal is to define optional predictable semantics for areas of C that are presently undefined. Various cases of undefined behavior in the standard could effectively be assigned such optional predictable semantics (at some optimization cost), while some changes from the present unbounded undefined behavior pose serious implementation problems in practice.

3.2.1 Division by 0, modulo 0 (signed or unsigned)

There is no defined infinite-precision result for division by zero. Processors typically trap, but this may not be universal. The AIR integer model requires trapping.

3.2.2 INT_MIN / -1

Processors commonly trap, but this may not be universal. For example, when using the IA-32 `idiv` instruction, dividing `INT_MIN` by `-1` results in a division error and generates an interrupt on vector 0 because the signed result (quotient) is too large for the destination [Intel 2004]. The AIR integer model requires trapping.

3.2.3 INT_MIN % -1

The ISO/IEC JTC1/SC22/WG14 C standards committee discussed the behavior of `INT_MIN % -1` on the WG14 reflector and at the April 2009 Markham meeting [Hedquist 2009]. The committee agreed that, mathematically, `INT_MIN % -1` equals 0. However, instead of producing the mathematically correct result, some architectures may trap. For example, implementations targeting the IA-32 architecture use the `idiv` instruction to determine the remainder. Consequently, `INT_MIN % -1` results in a division error and generates an interrupt on vector 0.

At Markham, some committee members argued that C99 requires a C program computing `INT_MIN % -1` to produce `0`, because `0` is representable while others argued that C99 left the computation as undefined because `INT_MIN / -1` is not representable. The committee decided that requiring C programs to produce `0` would render some compilers noncompliant with the standard, and that adding this corner case could add a significant overhead. Consequently, the C1X draft is being amended to explicitly state that if `a/b` is not representable, `a%b` is undefined.

The AIR integer model requires that `a % -1` equals `0` for all values of `a`, or alternatively, trapping is performed.

3.2.4 Shifts

Shifting by a negative number of bits or by more bits than exist in the operand is undefined behavior in C99 and, in almost every case, indicates a bug (logic error). Signed left shifts of negative values or where the result of the operation is not representable in the type are undefined in C99 and implementation-defined in C90. GCC has no options to handle shifts by negative amounts or by amounts outside the width of the type predictably or trap on them; they are always treated as undefined. Processors may reduce the shift amount modulo some quantity larger than the width of the type. For example, 32-bit shifts are implemented using the following instructions on IA-32:

```
sa[rl]l   %cl, %eax
```

The `sa[rl]l` instructions take a bit mask of the least significant 5 bits from `%cl` to produce a value in the range [0, 31] and then shift `%eax` that many bits.

64-bit shifts become

```
sh[rl]dl  %eax, %edx
sa[rl]l   %cl, %eax
```

where `%eax` stores the least significant bits in the double word to be shifted and `%edx` stores the most significant bits.

In the AIR integer model, shifts by negative amounts or amounts outside the width of the type trap because the results are not representable without overflow. This is consistent with The CERT C Secure Coding Standard Rule INT34-C, "Do not shift a negative number of bits or more bits than exist in the operand."

Signed left shifts of negative values, or where the result of the operation is not representable in the type, are undefined in C99 and implementation-defined in C90. In the AIR integer model, signed left shifts on negative values must not trap if the result is representable without overflow. If the value is not representable in the type, the implementation must trap. For example, `a << b == a * 2 ^ b` if `b >= 0` and `a * 2 ^ b` is representable without overflow in the type. For right shift, `a >> b == a / 2 ^ b` if `b >= 0`, and `2 ^ b` is representable without overflow in the type.

Unsigned left shifts never trap under the AIR integer model. This is because unsigned left shifts are generally perceived by programmers as loosing data, and there is a large amount of existing code that assumes modulo behavior. For example, in the following code from JasPer Version 1.900.1 `tmpval` is having `uint_fast32_t` type:

```
while (--n >= 0) {
  c = (tmpval >> 24) & 0xff;
  if (jas_stream_putc(out, c) == EOF) {
    return -1;
  }
  tmpval = (tmpval << 8) & 0xffffffff;
}
```

The modulo behavior of `tmpval` is assumed in the left shift operation.

3.2.5 Enabling and Disabling Unsigned Integer Wrapping

The default behavior under the AIR integer model is to trap unsigned integer wrapping.

Unsigned integer semantics are problematic because unsigned integer wrapping poses a significant security risk but is well-defined by the C standard. Also, in legacy code, the wrapping behavior can be critical to correct behavior. Consequently, it is necessary to provide mechanisms to enable and disable wrapping for unsigned integers.

Unsigned integer wrapping can be enabled or disabled per compilation unit or for individual variables.

Compiler options can be provided to enable or disable wrapping for all unsigned integer variables per compilation unit. Existing code that depends on modulo behavior for unsigned integers should be isolated in a separate compilation unit and compiled wrapping disabled.

When an unsigned integer defined in one compilation unit compiled with wrapping semantics is combined with another unsigned integer defined in a separate compilation unit with trapping semantics, the resulting value has the default behavior of the compilation unit in which the operation occurs.

New identifiers, such as `__wrap` and `__trap`, can be used as named attributes to enable or disable wrapping for individual integer variables, both signed and unsigned. These can be implemented as variable attributes in GCC or using `__declspec` or a similar mechanism in Microsoft Visual Studio until a standard attribute mechanism is specified by the C standards committee.

Enabling or disabling wrapping and trapping per variable has implications for the type system (what happens when you combine a wrapping variable with a trapping variable?) and for type safety (what happens when you pass a trapping variable as an argument to a function that accepts a wrapping parameter?). Because of these added complications, we recommend only allowing unsigned integer to be enabled or disabled per compilation unit.

Because of the sheer number of unsigned integer bugs seen by the security community, we strongly recommend that the trap behavior should be the default for all new code, and for as much legacy code as possible, consistent with adequate testing and code-review.

3.2.6 Integer Promotions and the Usual Arithmetic Conversions

In cases where a compilation unit is compiled with wrapping disabled for unsigned integers, it is possible that operations can take place between signed integers with trapping semantics and unsigned integers with wrapping semantics. In these cases, the semantics of the resulting variable (trapping or wrapping) depends on the integer promotions and the usual arithmetic conversions defined by C99. In cases where the resulting variable is a signed integer type, trapping semantics apply; in cases where the resulting value is an unsigned integer type, wrapping semantics are used.

3.2.7 Integer Constants

Expressions involving integer constants have the same semantics as similar expressions involving integer variables. The current C standard allows integer expressions to be reordered. For each of these expressions, there are a fixed number of possible orderings. For constant expressions, it is the compiler's responsibility to try all of the orderings. If one or more of these reordering evaluates the constant without overflowing, the implementation must not overflow in the calculation of the constant.

As implied by the name "as-if infinitely ranged," signed integer constant expressions are permitted to be evaluated as one of the greatest-width integer types, either `intmax_t` or `uintmax_t`, so overflow might not happen during the calculation of a constant; however, once the value is to be used or stored, that value must be within the correct range for its type.

We propose a more restricted model for trapping unsigned behavior where:

```
((unsigned)0 - 1)
```

produces a constraint error violation and should result in a fatal diagnostic if compiled.

Integer constants (literals) have the types as defined by C99 Section 6.4.4.1, "Integer constants." Signed integer constants under the AIR integer model are trapping. Under the default behavior of the model, unsigned integers constants are also trapping, unless explicitly disabled. For example, in the following expression all integer constants are of type `int`:

```
2 - 3 + 5
```

The compiler is required to try all orderings to find one that does not overflow. A valid ordering is:

```
2 + 5 - 3
```

The result is `3` (and not a constraint violation).

3.2.8 Expressions Involving Integer Variables and Constants

Because of macro expansion, another common case in C programs are expressions that include some number of variables and some number of constant values such as

```
V1 + C1 + V2 + C2
```

In this case, the compiler can reorder the expressions and reduce to a single constant value. However, the more restricted model for trapping unsigned behavior does not apply in this case.

For example, although the expression `1u - 2u` is a constraint violation if compiled with unsigned integer trapping, the following expression:

```
V1 + 1u + V2 - 2u;
```

can be reordered to

```
V1 + V2 + 1u - 2u
```

and consequently simplified to

```
V1 + V2 - 1u
```

Regardless of whether it is compiled with trapping enabled or disabled for unsigned integer values.

3.2.9 Runtime-constraint Handling

Most functions defined by ISO/IEC TR 24731-1-20074 include as part of their specification a list of runtime constraints, violations of which can be consistently handled at runtime [ISO/IEC TR 24731-1:2007]. Library implementations must verify that the runtime constraints for a function are not violated by the program. If a runtime constraint is violated, the runtime constraint handler currently registered with `set_constraint_handler_s()` is called.

Implementations are free to detect any case of undefined behavior and treat it as a runtime-constraint violation by calling the runtime-constraint handler. This license comes directly from

the definition of undefined behavior. Consequently, the AIR implementation uses the runtime-constraint mechanisms defined by ISO/IEC TR 24731-1-2007 for handling integer exceptional conditions.

Constraint handlers are defined to be of type `constraint_handler_t`. It has the following definition:

```
typedef void (*constraint_handler_t)(
  const char * restrict msg,
  void * restrict ptr,
  errno_t error
);
```

When the handler is called, it is passed the following arguments in the following order:

1. A pointer to a character string describing the runtime constraint violation.
2. A null pointer or a pointer to an implementation-defined object.
3. If the function calling the handler has a return type declared as `errno_t`, the return value of the function is passed. Otherwise, a positive value of type `errno_t` is passed.

The runtime constraint handler might not return.

3.3 Array Dereferences

An integer overflow during an array dereference constitutes an array out-of-bounds condition and is handled by the safe secure C/C++ (SSCC) methods in the analyzable model [Plum 05].

3.4 The rsize_t Type

The `rsize_t` type, defined as part of bounds-checked library functions [ISO/IEC TR 24731-1:2007], can be used in a complimentary fashion to AIR integers and is consequently subsumed as part of the overall solution. Functions that accept parameters of type `rsize_t` diagnose a constraint violation if the values of those parameters are greater than `RSIZE_MAX`. Extremely large object sizes are frequently a sign that an object's size was calculated incorrectly. For example, negative numbers appear as very large positive numbers when converted to an unsigned type like `size_t`. For those reasons, it is sometimes beneficial to restrict the range of object sizes to detect errors. For machines with large address spaces, ISO/IEC TR 24731-1 recommends that `RSIZE_MAX` be defined as the smaller of the size of the largest object supported or `(SIZE_MAX >> 1)`, even if this limit is smaller than the size of some legitimate, but very large, objects. The CERT C Secure Coding Standard [Seacord 2008] recommends using rsize_t or size_t for all integer values representing the size of an object (INT01-C).

3.5 Fussy Overflows

One problem with trapping is *fussy overflows*, which are overflows in intermediate computations that do not affect the resulting value. For example, on two's complement architectures, the following code

```
int x = /* nondeterministic value */;
x = x + 100 - 1000;
```

overflows for values of `x > INT_MAX - 100`, but underflow during the subsequent subtraction resulting in a correct as-if infinitely ranged integer value.

In this case, it is likely that most compilers will perform constant folding to simplify the above expression to `x - 900`, eliminating the possibility of a fussy overflow. However, there are situations where this will not be possible, for example:

```
int x = /* nondeterministic value */;
int y = /* nondeterministic value */;
x = x + 100 - y;
```

Because this expression cannot be optimized, a fussy overflow may result in a trap, and a potentially successful operation converted into an error condition.

3.6 Optimizations

An important consideration in adopting a new integer model is the effect on compiler optimization and vice versa. C language experts are accustomed to evaluating the CPU cost of various proposals. The usual comparison for the classic standards-conforming mode is to compare the CPU cost of solving the problem in the compiler versus the (zero) cost of not doing so. We submit that, for the analyzable mode, the proper comparison is to compare the CPU cost of analyzable generated code versus the CPU cost of the programmer's extra program logic added to the intrinsic CPU cost of the optimized construct. This comparison justifies putting a greater burden on the compiler when compiling otherwise insecure constructs in analyzable mode.

Regardless, performance is always an issue when evaluating new models and it is important to preserve existing optimizations while discovering new ones. Consequently, the AIR integer model does not prohibit any optimizations that are permitted by the C standard, but does require a diagnostic any time that the compiler performs an optimization based on a) signed integer overflow wrapping, b) unsigned integer wrapping, c) signed integer overflow cannot occur although value-range analysis cannot guarantee it will not, or d) unsigned integer wrapping cannot occur although value-range analysis cannot guarantee it will not.

For example, AIR integers allow optimizations based on algebraic simplification without a diagnostic:

```
(signed) (a * 10) / 10
```

This can be optimized to `a`. There is no need to preserve the possibly of trapping `a * 10`.

The expression

```
(a - 10) + (b - 10)
```

can be optimized to `(a + b) - 20`

While there is a possibility that `(a + b)` will produce a trap, there is also a possibility that either `(a - 10)` or `(b - 10)` would result in a trap in the original expression. So long as the application can be sure that each output is correctly represented, there is little interest in knowing whether a trap might have occurred by a different strategy.

Optimizations that assume integer overflow does not trap require a diagnostic because this assumption is inconsistent with the integer model. For example, certain optimizations operate on the basis that a loop must terminate by exactly reaching the limit `n`, and therefore the number of iterations can be determined by an exact division with no remainder (`EXACT_DIV_EXPR in GCC terms`) such as

```
for (i = 0; i != n; i += 3)
```

This loop can be optimized to iterate for some number of terms determined by a code sequence that is only valid for exact division and not if `n/3` leaves a remainder. This loop should also be diagnosed because it violates MSC21-C, "Use inequality to terminate a loop whose counter changes by more than one," in the CERT C Secure Coding Standard [CERT 2009a].

Diagnostics are also required for optimizations on pointer arithmetic that assume wrapping cannot occur.

3.7 GCC Prototype

A proof of concept modification to GCC compiler version 4.5.0 has been developed by the CERT Secure Coding Initiative that automatically invokes a runtime constraint handler when an integral operation fails to produce a correctly represented value.

To diagnose integer overflow properly on an x86 processor, it is necessary to know whether the arguments are signed or unsigned, so that the appropriate flag (carry or overflow) can be checked. The overflow flag indicates overflow has occurred for signed operations while the carry flag can be safely ignored. For unsigned computations, the opposite is true. Unfortunately, GCC's last internal representation, the register transfer language (RTL), has no way of storing the signedness of arguments to operations.

This requires inserting a flag into the RTL data structure, the `rtx`, which carries signedness information to the GCC backend where translation to assembly code is performed. Upon translation, the proper signedness information is available to produce the correct RTL pattern.

Conditional jumps are added to RTL patterns containing arithmetic operations to invoke a runtime constraint handler in the event that signed overflow or unsigned wrapping occurred, as shown in the following example:

```
   // arithmetic
jn[co] .LANALYZEXXX
   call constraint_handler

.LANALYZEXXX
   // code after arithmetic
```

Overflow checks were not added following signed division because these operations result in a division error on IA-32 and generate an interrupt on vector 0.

The performance of the prototype was assessed using the industry standard SPECCPU2006 benchmarks, which provide a meaningful and unbiased metric. Because the goal of this project is analyzable integer behavior, only the SPECINT2006 portion of the benchmark was run.

Because the benchmarks used in SPECINT2006 are not designed for analyzable code, the prototype was slightly modified so that the analysis is performed, but programs do not abort in the case of an overflow. The new snippet has a `nop` instruction instead of a call to a runtime constraint handler:

```
   jn[co] .LANALYZEXXX
   nop
.LANALYZEXXX
```

The prototype was slightly modified so that calls to constraint handlers were replaced by `nop` instructions. This modification prevents runtime constraint handlers from being invoked in the case of overflow and wrapping behavior in the benchmarks.

As explained in Section 2.5, the higher numbers in Table 4 indicate better performance.

Table 4. SPECINT2006 macro-benchmark runs

Optimization Level	Control Ratio	Analyzable Ratio	% Slowdown
-O0	4.92	4.60	6.96
-O1	7.21	6.77	6.50
-O2	7.38	6.99	5.58

Code insertions occur after all optimizations are performed by GCC, so the observed slowdown is not caused by disrupted optimizations. Instead, the slowdown is entirely due to the cost of the conditional tests after each arithmetic operation.

Runtime performance could be further improved by eliminating unnecessary tests in cases where value-range analysis can prove that overflow or wrapping is not possible.

3.8 Experimental Results

The effectiveness of the GCC prototype was evaluated by compiling the JasPer software (jasper-1.900.1) and running the resulting executable with some actual data. Fuzz testing is planned but has yet to be performed.

JasPer includes a software-based implementation of the codec specified in the JPEG-2000 Part-1 standard ISO/IEC 15444-1 and is written in the C programming language. The JasPer software has been included in the JPEG-2000 Part-5 standard ISO/IEC 15444-5, as an official reference implementation of the JPEG-2000 Part-1 codec. This software has also been incorporated into numerous other software projects (some commercial and some non-commercial). Some projects known to use JasPer include

- K Desktop Environment (as of version 3.2)
- Kopete
- Ghostscript
- ImageMagick
- Netpbm (as of Release 10.12)
- Checkmark

Because this library has been included in many commercial and non-commercial applications that have been widely deployed and used, any vulnerabilities in this library are quite severe because they can lead to wide spread compromises.

Our initial assessment of JasPer using GCC discovered one potential vulnerability and two false positives. The vulnerability involved wrapping of an unsigned integer value. Further details will be published pending completion of a responsible vulnerability disclosure process.

The two false positives both involved signed left shift. The first false positive occurs in `src/libjasper/bmp/bmp_dec.c:443` in function `bmp_getint32()`

```
static int bmp_getint32(jas_stream_t *in, int_fast32_t *val) {
  int n;
  uint_fast32_t v;
  int c;
  for (n = 4, v = 0;;) {
    if ((c = jas_stream_getc(in)) == EOF) {
      return -1;
    }
    v |= (c << 24);
    if (--n <= 0) {
      break;
    }
    v >>= 8;
  }
  if (val) {
    *val = v;
  }
  return 0;
}
```

The `jas_stream_getc()` function has similar semantics to `getc()`, meaning it returns `EOF` or an `unsigned char` cast to an `int`. Consequently, the left shift 24 of `c` is safe provided that the platform is non-trapping, although it causes a signed overflow and undefined behavior.

The false positive can be eliminated by casting **`c`** to an unsigned integer type as follows:

```
    v |= ((unsigned int)c << 24);
```

While this is a false positive for a vulnerability, it is undefined behavior and is also a violation of "INT13-C. Use bitwise operators only on unsigned operands" in the CERT C Secure Coding Standard [Seacord 2008].

The second false positive is similar:

```
static int ras_putint(jas_stream_t *out, int val) {
  int x;
  int i;
  int c;

  x = val;
  for (i = 0; i < 4; i++) {
    c = (x >> 24) & 0xff;
    if (jas_stream_putc(out, c) == EOF) {
      return -1;
    }
    x <<= 8;
  }
  return 0;
}
```

The problem here is also the left shift of a signed integer, which can be eliminated by casting the signed integer to a compatible unsigned integer type.

Summary and Conclusions

Previous attempts to trap overflows created runtime overhead and hampered optimization. However, there is considerable latitude for optimization within the C and C++ standards that has not been exploited.

The AIR integer model guarantees correctly represented integers or trapping, at minimum overhead. Further work is planned to evaluate the security of applications with and without the use of AIR integers.

We understand that a fully complete implementation should be portable to non-x86 architectures as well as language independent.

The current implementation is easily extensible to achieve these goals on architectures which support overflow checks based on signedness. The RTL representation is both architecture and language independent, so no changes would be needed before or during the generation of the RTL representation. However, to take advantage of the signedness information present in the RTL representation, each targeted architecture would need to have its own patterns modified.

It is likely that such an extension would represent a significant undertaking for those involved, as the history of GCC has a led to a very large and complicated code base. We also hope that similar functionality could be implemented in other leading C compilers, including LLVM/Clang, the Intel C/C++ Compiler, and Microsoft Visual C++.

We also hope that in the future, integer truncation errors can be analyzed and caught efficiently. However, because on most architectures truncation can be analyzed in software, for the time being, we expect programmers to manually assert the safety of their type conversion and truncation operations.

References/Bibliography

URLs are valid as of the publication date of this document.

[Adams 2006]
Michael D. Adams. *JasPer Software Reference Manual* (Version 1.900.0). December 2006. http://www.ece.uvic.ca/~mdadams/jasper/jasper.pdf

[Brumley 2007]
Brumley, David; Chiueh, Tzi-cker; Johnson, Robert; Lin, Huijia; & Song, Dawn. "RICH: Automatically Protecting Against Integer-Based Vulnerabilities." *Proceedings of NDSS*, San Diego, CA, Feb. 2007. Reston, VA: ISOC, 2007. http://www.cs.berkeley.edu/~dawnsong/papers/efficient_detection_integer-based_attacks.pdf

[CERT 2009a]
CERT. *CERT Secure Coding Standards*. Pittsburgh, PA. CERT, March 2009. https://www.securecoding.cert.org/confluence/x/BgE

[CERT 2009b]
The CERT C++ Secure Coding Standard. Pittsburgh, PA. CERT, March 2009. https://www.securecoding.cert.org/confluence/x/fQI

[Christey 2007]
Christy, Steve & Martin, Robert A. Vulnerability Type Distributions in CVE. Document Version: 1.1. May 22, 2007. http://cve.mitre.org/docs/vuln-trends/vuln-trends.pdf

[CVE 2001]
CVE. "CVE-2001-0144, SSH CRC-32 compensation attack detector vulnerability," Bedford, MA: Mitre Corp., Feb 2001. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2001-0144

[Hedquist 2009]
Hedquist, Barry. *ISO/JTC1/SC22/WG14 AND INCITS PL22.11 September 2008 Meeting Minutes (Draft)*. Washington, D.C.: InterNational Committee for Information Technology Standards, 2008. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1380.pdf

[Heffley 04]
Heffley, Jon & Meunier, Pascal. "Can Source Code Auditing Software Identify Common Vulnerabilities and Be Used to Evaluate Software Security?" *Abstracts Proceedings of the 37th Hawaii International Conference on System Sciences (HICSS-37 2004)* (CD-ROM). Big Island, HI, January 5-8 2004. Washington, D.C.: IEEE Computer Society, 2004.

[Intel 2004]
Intel, Corp. *The IA-32 Intel Architecture Software Developer's Manual*. Denver, CO: Intel Corp., 2004.

[ISO/IEC TR 24731-1:2007]
International Standards Organization. *ISO/IEC TR 24731. Extensions to the C Library, — Part I: Bounds-checking interfaces*. Geneva, Switzerland: International Standards Organization, April 2006.

[Meyers 2003]
Meyers, Randy & Plum, Thomas. "The 20th Anniversary of the C Standards Committee: Moore's Law and then some--how far we've come!" *C/C++ Users Journal*. September, 2003. http://www.ddj.com/cpp/184401697

[Plum 05]
Plum, Thomas & Keaton, David M. "Eliminating Buffer Overflows, Using the Compiler or a Standalone Tool." *Proceedings of the Workshop on Software Security Assurance Tools, Techniques, and Metrics*, Long Beach, CA, November 7-8, 2005. Washington, D.C.: U.S. National Institute of Standards and Technology (NIST), 2005. http://samate.nist.gov/docs/NIST_Special_Publication_500-265.pdf

[Plum 09]
Plum, Thomas. & Seacord, Robert C. *ISO/IEC JTC 1/SC 22/WG14/N1350 – Analyzability*. Geneva, Switzerland: International Standards Organization, February 2009. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1350.htm

[Saltzer 75]
Saltzer, Jerome H. & Schroeder, Michael D. "The Protection of Information in Computer Systems," 1278-1308. *Proceedings of the IEEE 63*, 9 (September 1975).

[Seacord 2005]
Seacord, Robert C. *Secure Coding in C and C++*. Boston, MA: Addison-Wesley Professional, 2005.

[Seacord 2008]
Seacord, Robert. *The CERT C Secure Coding Standard*. Boston, MA: Addison-Wesley, 2008.

[Solar Designer 00]
Solar Designer. *JPEG COM Marker Processing Vulnerability in Netscape Browsers*. OpenWall Project, July, 2000. http://www.openwall.com/advisories/OW-002-netscape-jpeg.txt

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	July 2009	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
As-if Infinitely Ranged Integer Model	FA8721-05-C-0003

6. AUTHOR(S)

David Keaton, Thomas Plum, Robert C. Seacord, David Svoboda, Alex Volkovitsky, Timothy Wilson

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2009-TN-023

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

Integer overflow and wraparound are a major cause of software vulnerabilities in the C and C++ programming languages. In this paper, we present the as-if infinitely ranged (AIR) integer model, which provides a largely automated mechanism for eliminating integer overflow and integer truncation. The AIR integer model either produces a value which is equivalent to a value that would have been obtained using infinitely ranged integers or results in a runtime constraint violation. Unlike previous integer models, AIR integers do not require precise traps, and consequently do not break or inhibit most existing optimizations.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Infinitely ranged integers, software security, secure coding	34

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18
298-102